LA META-EVALUATION AU SERVICE DE LA COMPREHENSION DE PROGRAMMES

Daniel GOOSSENS

Le programmeur expert qui écrit, étend ou modifie en parallèle plusieurs programmes participant tous d'un système unique, aura besoin d'un assistant qui ***dénonce*** interactivement les imperfections issues de son incapacité à contrôler l'organisation globale de ce système.

C'est à étudier comment atteindre ce but qu'est consacré le système CAN, un système de compréhension automatique de programmes LISP, présenté dans cette étude. CAN est implémenté en VLISP [GREUSSAY 76 77 78 79, CHAILLOUX 78a 78b 78c] et tourne sur un PDP-10 modèle KI-10. Il y occupe 20K mots de 36 bits.
CAN a la capacité de déceler dans un système dont il ne connait pas l'intention des particularités telles que :

- portions de code non utilisées
- classes de données pour lesquelles le calcul ne termine pas
- redondances
- effets de bord indésirables

La compréhension dont CAN est capable est fondée sur un processus compact de ***méta-évaluation***. La méta-évaluation permet d'effectuer en un seul calcul symbolique une infinité de calculs particuliers. La méta-évaluation utilisée par CAN représente les connaissances au sujet de programmes sous forme de systèmes d'équations et se charge de les résoudre. Elle sert à analyser dans un programme le flot des données, dans le but d'en exhiber des particularités (calculs infinis, portions de code non utilisées, chemins impossibles, redondances). La méta-évaluation utilisée par le système CAN répond à deux types de problèmes posés par l'évaluation symbolique:

- L'évaluation symbolique, dans ses réalisations actuelles [LONDON 74, KING 75 76, BOYER 75b, YONEZAWA 75 76, BALZER 77, CHEATHAM 79] ne permet pas que soit décrit de façon symbolique autre chose que les valeurs associées aux identificateurs, dans un environnement. De ce fait, elle ne traite pas de nombreuses constructions, fréquement utilisées en LISP: variables fonctionelles, macro-génération dynamique de fonctions incomplètement spécifiées, définition de structures de contrôle.

- Les résultats fournis par l'évaluateur symbolique sont souvent aussi peu transparents que le programme évalué symboliquement, surtout en ce qui concerne les programmes de traitement de listes. Les représentations traditionellement utilisées par l'évaluation symbolique, les formules de la logique des prédicats, n'exhibent pas plus que les programmes qu'elles annotent, les redondances, les classes de données pour lesquelles le programme ne termine pas son calcul, les propriétés <u>utiles</u> de programmes.

La META-EVALUATION utilisée par le système CAN augmente l'évaluation symbolique dans ces deux directions:

- Elle l'étend aux constructions nouvelles permises par LISP: style applicatif, macro-génération de programmes, affectations où l'identificateur n'est pas explicité, structures de contrôle définies par l'utilisateur, appels à l'évaluateur, programmation incrémentale, entrées-sorties.

- Elle repose sur un nouveau mode de représentation des données symboliques: les ***représentations conceptuelles*** [YONEZAWA 76]. CAN est capable d'associer à un programme une représentation conceptuelle de son activité et de ses propriétés principales. Cette traduction en représentations conceptuelles a pour but d'isoler les problèmes de compréhension au sujet d'un complexe de programmes, sous forme d'***équations*** à résoudre. CAN utilise alors des systèmes indépendants de résolution d'équations, spécialisés en particulier dans le traitement de listes.

La méta-évaluation doit être utilisée et contrôlée par des systèmes plus larges de ***compréhension automatique*** de programmes.

Considéré comme un système de compréhension de programmes LISP, CAN associe à un programme une définition sémantique, qui lui est équivalente, mais de laquelle il est beaucoup plus aisé d'extraire des propriétés importantes. Les langages de programmation, et LISP en particulier, déploient une immense variété de représentations de calculs, et permettent d'oblitérer des propriétés simples derrière des formules courtes, mais à la sémantique complexe.
Pour CAN, comprendre un programme, c'est y associer sa représentation conceptuelle. Les représentations conceptuelles forcent l'extraction de ce qui est habituellement caché dans un programme: effets de bord, contrôle de l'évaluation, analyses de cas, structure dymamique des objets manipulés par le programme. Elle peuvent ainsi isoler des particularités interessantes d'un programme, telles que:

- effets de bord imprévus
- classes de données pour lesquelles le calcul du programme ne termine pas
- redondances
- portions de code non utilisées.

L'EVALUATION SYMBOLIQUE

Il est possible de modifier un interprète pour qu'il tienne compte de la présence de ***valeurs abstraites*** dans l'environnement [note 1] d'une expression à évaluer. Evaluer symboliquement une expression, c'est l'évaluer dans un environnement insuffisamment décrit pour pouvoir utiliser un évaluateur classique. Evaluer symboliquement cette expression consiste à compléter la description de l'environnement sur la seule base des exigences de cette expression, et puis évaluer cette

expression normalement.

LA META-EVALUATION

Le domaine d'entrée de l'évaluateur EVAL du langage LISP est l'ensemble des couples <EXPRESSION , ENVIRONNEMENT> où EXPRESSION est une expression LISP, et où ENVIRONNEMENT donne accès à une liste de couples <IDENTIFICATEUR , VALEUR>.

Le domaine d'entrée du méta-évaluateur est l'ensemble des couples *<a,b>* où *a* et *b* sont respectivement une expression et un environnement méta-décrits.

L'évaluateur symbolique admet que VALEUR soit méta-décrit, mais s'attend à ce que IDENTIFICATEUR et EXPRESSION soient des données concrètes.
l'examen du langage LISP montre qu'il est nécéssaire que l'évaluateur symbolique s'attende à ce que IDENTIFICATEUR et EXPRESSION soient méta-décrits. Il doit être étendu en un méta-évaluateur.

Exemple :

Le langage LISP autorise l'usage d'arguments fonctionnels. Comprendre un programme qui possède des arguments fonctionnels, c'est pouvoir simuler l'application sur des données d'un programme insuffisamment décrit, c'est-à-dire un schéma de programmes.

Le langage LISP permet, par exemple, qu'en position de fonction, dans une expression non atomique, se trouve une autre expression à évaluer, censée s'évaluer en une fonction qui sera alors appliquée sur les arguments.

On peut écrire en LISP des programmes qui construisent d'autres programmes à partir de données, et qui les utilisent.

L'exemple suivant, simplifié, est tiré du générateur de conditions à vérifier de [IGARASHI 73] programmé en VLISP.

```
(DE FOO (a b)
    [λ [a]
       ['SUBSTITUER
        ['QUOTE a]
        ['QUOTE b]]])

(DE SUBSTITUER (s l)
```

[note 1] Pour l'évaluateur symbolique présenté dans ce chapitre, l'environnement est une liste de couples (identifieur,valeur). Le mode de définition de l'évaluateur symbolique laisse cependant la possibilité de détailler d'autres structures dans l'environnement. C'est ce qui est fait au chapitre 4, où l'on y introduit une pile pour les fonctions d'échappement, et des tampons d'entrée et de sortie.

```
(SUBST (EVAL s) s l))
```

Dans un contexte créé par un appel de FOO, SUBSTITUER remplace dans la liste contenue dans l toutes les occurences de l'atome contenu dans s par la valeur de cet atome dans l'environnement courant.
L'erreur contenue dans cet exemple est de même nature que celle contenue dans le programme FILTRER, exemple du paragraphe 4.1.1, quoique plus complexe. Les méthodes classiques de test sur exemples précis ne marchent donc pas plus.

Le résultat d'un appel de FOO est utilisé comme un programme et appliqué sur une valeur quelconque. Voici le comportement souhaité de l'évaluateur symbolique:

<((FOO '*a* '*b*) '*v*) , {}>

commentaire: le travail de FOO n'est pas détaillé. Une fois sa valeur retournée, les liaisons de ses paramètres formels sont détruites.

<((λ (*a*) (SUBSTITUER (QUOTE *a*) (QUOTE *b*))) '*v*) , {}>

commentaire: L'évaluateur symbolique doit utiliser ici des connaissances sur la syntaxe des λ-expressions et se polariser sur le cas où ***a*** est un atome litéral.

<(SUBSTITUER (QUOTE *a*) (QUOTE *b*)) , {*a*=*v*}>

<((λ (s l) (SUBST (EVAL s) s l)) (QUOTE *a*) (QUOTE *b*)) , {*a*=*v*}>

commentaire: pour affecter les valeurs symboliques ***a*** à s et ***b*** à l, l'évaluateur symbolique tient compte des possibilités:

a=s	*a*=l	*a*≠s, *a*≠l
<(SUBST (EVAL s) s l) , {s=s l=*b*}>	<(SUBST (EVAL s) s l) , {s=l l=*b*}>	<(SUBST (EVAL s) s l) , {s=*a* l=*b* *a*=*v*}>
(SUBST 's 's '*b*)	(SUBST '*b* 's '*b*)	(SUBST '*v* '*a* '*b*)

(SUBST x y z) substitue x à y dans z. Seule la troisième branche correspond à l'intention initiale, qui est de substituer dans ***b*** toute occurence de ***a*** par ***v***.

L'évaluation symbolique fournit les exemples pour lesquels l'intention n'est pas satisfaite:

1. ((FOO 's '*b*) '*v*)

2. ((FOO 'l '*b*) '*v*)

où s et l sont des atomes litéraux et *b* et *v* des valeurs abstraites.

LA RESOLUTION D'EQUATIONS

Les définitions sémantiques des fonctions de base ne décrivent plus les raisonnements à effectuer devant tous les types de situations rencontrables. Ces raisonnements sont isolés en systèmes indépendants. Les définitions sémantiques se contentent de décrire statiquement le domaine d'entrée, de sortie, la relation fonctionelle, ou tout autre aspect que l'on veut exhiber dans les résultats d'une méta-évaluation.

La fonction META-APPLY devient un ***poseur d'equations.*** Si par exemple la définition sémantique d'une fonction de base ***FCT*** exhibe une description ***<l1 , env1>*** de son domaine d'entrée,

(META-APPLY '***FCT l2 env2***)

entre autres, pose l'équation:

<l1 , env1> = <l2 , env2>

C'est le système de raisonnement, indépendant du langage de programmation, qui se charge de résoudre cette équation.

exemple:
considérons l'exemple 1 du test LISP :

(LAST (APPEND L M))

Lors de la méta-évaluation, lorsque la méta-valeur de la sous-expression (APPEND L M) est calculée :

<***(append l m)*** , {L = *l*,M = *m*}>

la fonction LAST y est méta-appliquée:

(META-APPLY 'LAST '***(append l m)*** {L = *l*,M = *m*})

cet appel pose tour à tour les équations:

1- <***(append l' (cons x nil))*** , ***env***>
= <***(append l m)*** , {L = *l*,M = *m*}> (1)

2- <***()*** , ***env***>

= <*(append l m)* , {L = *l*,M = *m*}>

les définitions sémantiques des fonctions META-CAR et META-LAST:

META-LAST:

1- <*(append l' (cons x nil))* , *env*> → <*x* , *env*>

2- <*()* , *env*> → <*()* , *env*>

META-CAR:

1- <*(cons x l')* , *env*> → <*x* , *env*>

2- <*()* , *env*> → <*()* , *env*>

sont courtes et non redondantes. Les raisonnements communs à leur deux défintions procédurales sont isolés dans le système de raisonnement par résolution d'équations:

le raisonnement commun à la méta-évaluation des 2 formes (LAST (APPEND L M)) et (CAR (APPEND L M)) est la résolution de l'équation :

(append a b) = (append c d)

Pour l'exemple présent, les équations ① se résolvent ainsi:

1- ***(append l' (cons x nil)) = (append l m)***

donne deux solutions:

- ***m = (append m' (cons x nil))***
  l' = (append l m')

- ***m = ()***
  l = (append l' (cons x nil))

2- ***() = (append l m)***

donne 1 solution:

- ***l = ()***
  m = ()

Ces trois solutions fournissent le résultat escompté.

UN SYSTEME DE REPRESENTATION DE SIGNIFICATIONS

Un système de compréhension automatique de programmes ne doit pas se contenter de décrire symboliquement la succession des états de la machine durant l'évaluation de l'appel d'un programme. Il doit fournir une ***signification*** de ce programme, objet formel à partir duquel les applications de la compréhension (diagnostics d'erreurs, extraction de propriétés utiles, annotations, vérifications, corrections) doivent être facilitées. Les représentations utilisées par le méta-évaluateur défini au chapitre 4, les listes de couples <valeur, environnement>, sont insuffisantes pour représenter des programmes LISP.

Voici un système de représentation de significations de programmes. Ces représentations sont indépendantes d'un domaine de données (par exemple le domaine des listes, en LISP, sur lequel sont définies les fonctions CAR CONS CDR NULL). Elles concernent le contrôle de l'évaluation :

- conditionelles (IF COND AND OR SELECT)
- itération (WHILE UNTIL MAPC MAPCAR DO)
- appels à l'évaluateur (EVAL)
- échappement (ESCAPE EXIT)
- appel par nom ou par valeur (DE DF)

(les exemples sont ceux du langage VLISP)

Ces représentations tiennent compte également de :

- définitions de structures de contrôle (DF et EVAL)

- arguments fonctionnels, expressions incomplètement spécifiées (macro-génération dynamique de fonctions)

- appels récursifs

- entrées-sorties (READ PRINT)

- programmation incrémentale (appels à des programmes non encore définis)

Appliqué à une expression *A* et un environnement méta-décrit *env*, et s'il termine son calcul, le méta-évaluateur retourne une liste de couples *⟨valeur , environnement⟩* tels que l'ensemble des formes [*env* → *⟨valeur , environnement⟩*] décrit la fonction dénotée par *A*. Si *A* est le corps d'un programme (λ *PFS . A*), alors l'ensemble des formes [*⟨PFS , env⟩* → *⟨valeur , environnement⟩*], résultat de la méta-évaluation, est la définition sémantique associée à la λ-expression.

En appliquant le méta-évaluateur sur *A* dans l'environnement le plus global construit à partir de *PFS*, on peut obtenir une définition de la λ-expression, à laquelle le méta-évaluateur peut se référer lorsqu'il a à comprendre une expression qui comporte un appel de cette

λ-expression. Si une telle définition a été obtenue, le méta-évaluateur peut évaluer simplement une application de la λ-expression sur des arguments concrets, toujours en se référant à la définition sémantique.

exemple:

à partir de la λ-expression:

(λ (L M) (LAST (APPEND L M))) ①

le méta-évaluateur obtient:

<l (append m (cons x nil)) , env> → *<x , env>*

<(append l (cons x nil)) () , env> → *<x , env>*

<() () , env> → *<() , env>*

Ceci est un programme au même titre que ①. Le méta-évaluateur peut simplement évaluer les appels :

(① '(a b c) '(d e)) = e

(① '(a b c) '()) = c

(① '() '(a b)) = b

(① '() '()) = ()

L'examen de cas concrets conduit à un concept de ***point de rupture*** qui permet de représenter de façon uniforme des constructions aussi diverses que:

- appels à l'évaluateur (fonction LISP EVAL)

- définition de structures de contrôle (à l'aide des fonctions LISP DF et EVAL)

- arguments fonctionels, schémas de programmes

- programmes à appel par valeur ou par nom (fonctions LISP DE et DF)

- définitions récursives

- processus interactifs, non-terminants, à base d'entrées-sorties

- programmation incrémentale. Des programmes incomplets, faisant appel à d'autres programmes non encore définis, peuvent toutefois être compréhensibles.

Exemple :

Voici la définition sémantique de (l'application de) EVAL:

```
<(EVAL a) , {Li=vi}>
        → (point-de-rupture                    ②
            <a , {Li=vi}>
            <a' , {Li=vi'}> → <a' , {Li=vi'}>)
```

La forme (point-de-rupture *exp* . *regles*) indique que *exp* est une expression à méta-évaluer. *regles* est une continuation.

Exemple :

Les deux définitions :

```
(DE FOO1 (X) X)

(DF FOO2 (L) (EVAL (CAR L)))
```

sont presque équivalentes. C'est à dire que les appels :

```
(FOO1  E)
(FOO2  E)
```

où E est une expression LISP, fournissent la même valeur, par exemple :

```
(FOO1 (CAR '(a b c))) = a
(FOO2 (CAR '(a b c))) = a
```

à l'exception d'une classe de cas, dont les appels :

```
(FOO1 (CAR L))
(FOO2 (CAR L))
```

sont un exemple.
Si L = (a b c),

```
(FOO1 (CAR L)) = a
(FOO2 (CAR L)) = (CAR L)
```

FOO1 et FOO2 reçoivent respectivement les définitions (l'atome point-de-rupture est abbrévié à PR):

```
<(FOO1 EXP) , env>
     → (PR <EXP , env>
           <x , env'>
                → (PR <(LIER 'X 'x) , env'>
                      <anc-val , env''>
                           → (PR <(DELIER X anc-val) , env''>
                                 env''' → <x , env'''>)))
```

qui, par connaissance de LIER et DELIER, se simplifie en:

```
<(F001 EXP) , {L=anc-val}>
     → (PR <EXP , {L=anc-val}>
          <x , {L=v}> → <x , {L=anc-val}>)
```

et:

```
<(F002 EXP . Y) , env>
     → (PR <(LIER 'L '(EXP . Y)) , env>
          <anc-val , env'>
             → (PR <EXP , env'>
                  <x , env''>
                     → (PR <(DELIER L anc-val) , env''>
                          env''' → <x , env'''>)))
```

qui se simplifie en:

```
<(F002 EXP . Y) , {L=anc-val}>
     → (PR <EXP , {L=(EXP . Y)}>
          <x , {L=v}> → <x , {L=anc-val}>)
```

La différence entre F001 et F002 apparait à la forme (PR <EXP , *env*>. Pour F001, EXP est évalué dans l'environnement de départ. Pour F002, l'environnement contient la liaison supplémentaire L=(EXP . Y).
Ces deux définitions exhibent à la fois les points communs de F001 et F002 (l'évaluation de leur premier argument) et leurs différences (pour F002, cette évaluation se fait *aprés* la liaison du paramètre L).

Exemple :

CAN transforme le programme :

```
(DE FOO ()
   (F (READ))
   (FOO))
```

où F est un programme quelconque, en :

```
<(FOO) , env>
     → (PR <(READ) , env>
          <v , env'> → (PR <(F v) , env'>
                           <v' , env''>
                              → (PR <(FOO) , env''>
                                   v'' → v'')))
```

où *v*, *v'* et *v''* sont des variables dont le domaine est l'ensemble des éléments LISP.

LE CALCUL CONCEPTUEL

Le calcul conceptuel permet de représenter sous le même statut les définitions sémantiques des unités syntaxiques de base, dont se sert le méta-évaluateur, d'un langage de programmation.

La méta-évaluation, appliquée à ces unités syntaxiques:

- manipule plusieurs contextes,

- compare des données méta-décrites (résolution d'équations).

Ces deux fonctions sont construites à priori dans l'interprète du calcul conceptuel et n'ont pas besoin d'être représentés dans la syntaxe externe.

Ecrire un programme ou un schéma de programmes en calcul conceptuel permet de le tester sur des données méta-décrites, sans qu'il soit nécéssaire de modifier le programme. Le calcul conceptuel permet la ***méta-programmation.***

exemple:
Voici un concept qui définit la fonction LAST. LAST évalue son argument, s'attend à ce que la valeur soit une liste, retourne () si elle est vide, et son dernier élément sinon.

<(LAST L) , *env*>
→ (<u>point-de-rupture</u>
<L , *env*>
<() , *env'*> → <() , *env'*>
<*(append l (cons x nil))* , *env'*>
→ <*x* , *env'*>)

L'argument L de LAST peut être une expression quelconque. Le terme conceptuel <L , *env*>, premier argument du point de rupture, fait référence aux définitions conceptuelles présentes, s'il y en a. ***évaluer*** une forme (LAST expression) en calcul conceptuel, c'est donc d'abord ***évaluer*** expression, puis comparer les résultats obtenus avec chacun des termes conceptuels gauche des concepts du point de rupture: <() , *env'*> et <*(append l (cons x nil))* , *env'*>. Chaque concept fournit une valeur et un environnement résultant de l'évaluation de l'expression.

La règle de composition est la principale règle de ***simplification*** du calcul conceptuel. étant donnés deux concepts [t1 → t2] et [t3 → t4], où t1, t2, t3 et t4 sont des termes conceptuels, elle transforme la composition:

[t1 → t2]o[t3 → t4]

en un ensemble de concepts de la forme:
[E → F]

Si un concept est interprété comme une procédure à invocation par ***résolution d'équation***[note 1], alors le point de rupture peut être interprété comme:

(<u>point-de-rupture</u> appel-de-procédure . continuation)

La règle d'équilibration relie les concepts entre eux.

La règle du cercle vicieux :

- Du point de vue de la méta-évaluation, la règle du cercle-vicieux permet d'arrêter des calculs récursifs sur des données méta-décrites lorsqu'il est certain qu'aucune nouvelle information ne peut être obtenue.

- Du point de vue de la compréhension de programmes et de la construction de définitions à partir de programmes, la règle du cercle-vicieux permet de ne pas se contenter d'arrêter la méta-évaluation à chaque appel récursif, et de poursuivre lorsque l'appel récursif n'est pas *équivalent* à l'appel d'origine.

La règle d'induction constructrice est un rouage indispensable de la compréhension de programmes. Le système CAN obtient à partir de définitions récursives ou itératives des définitions conceptuelles dont il se sert pour effectuer des simplifications impossibles par simples pliages et dépliages des définitions récursives.

L'évaluation

Le calcul conceptuel permet d'évaluer un terme conceptuel de base dans un contexte formé d'un ensemble de concepts, de la même manière qu'il est possible, en LISP, d'évaluer une expression dans un contexte de définitions de fonctions.

Evaluer un terme de base t (sans variable) en calcul conceptuel, c'est appliquer la règle d'équilibration sur le concept:

```
[t → (PR t
         t' → t')]
```

où le concept [t' → t'] représente le fonction identité partout définie.

L'évaluation symbolique

Une évaluation intermédiaire entre cette évaluation et la méta-évaluation, l'évaluation symbolique, admet que le terme à évaluer soit un terme conceptuel quelconque. Ce terme est une donnée symbolique, ou méta-décrite. C'est la représentation symbolique d'une classe infinie de termes conceptuels de base.

La règle de composition doit résoudre des équations t=t' où t et t' sont des termes quelconques.

[note 1]: Les langages PLANNER [HEWITT 72] et CONNIVER [Mc.DERMOTT 72] ont introduit la notion de "pattern directed invocation", ou invocation par filtrage. La notion d'invocation par résolution d'équation en est une extension. Un exemple particulier d'invocation par résolution d'équation est l'invocation par unification en PROLOG [ROUSSEL 75].

Dans le cas où les termes conceptuels ne contiennent que des variables d'élément, la résolution d'équations se limite à l'unification robinsonienne [ROBINSON 65]. Les règles de composition et d'équilibration correspondent aux règles d'évaluation de clauses de HORN, en logique des prédicats, interprétées comme des procédures [VAN EMDEN 76]. La règle de résolution [ROBINSON 65], interprétée comme une règle d'invocation de procédure, correspond aux deux règles de composition et d'équilibration.

La méta-évaluation

Etant donné un concept interprété comme une forme à méta-évaluer, la règle d'équilibration peut en général s'appliquer de plusieurs manières. Elle peut s'appliquer sur le concept lui-même, et sur chacun de ses sous-concepts.
La méta-évaluation impose un choix simple pour l'application de la règle d'équilibration. Etant donnés un concept et un ensemble de choix d'application, la règle d'équilibration est appliquée sur le concept le plus extérieur. Si le concept n'est pas modifié par la règle d'équilibration, alors elle est réappliquée sur le second sous-concept le plus extérieur. S'il est modifié, le processus recommence. Ce processus termine lorsque la règle d'équilibration ne peut plus modifier le concept.

L'Analyse de programmes récursifs

Les buts d'une analyse de programmes récursifs, une fois représentés conceptuellement, sont de :

- décrire les classes de données pour lesquelles le calcul ne termine pas, ou le programme n'est pas défini.

- déceler les portions de code non utilisées, ou les chemins impossibles

- décrire de façon statique l'activité d'un programme.

- exhiber des inefficiences

- reconnaître l'équivalence sémantique profonde de deux programmes.

Exemple :

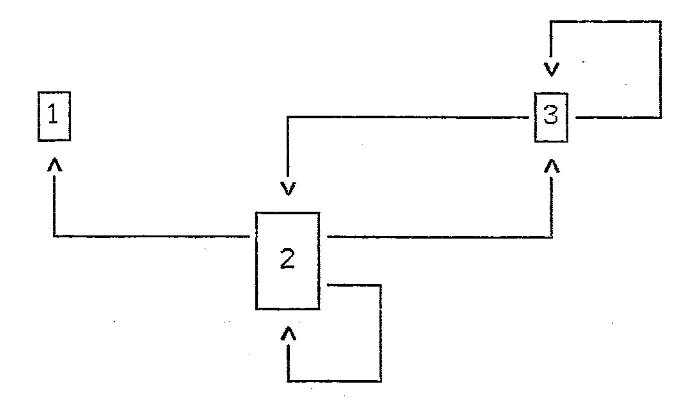
Au programme LISP qui concatène un nombre quelconque de listes :

```
(DE CONC (L)
    (COND ((NULL L) NIL)
          ((NULL (CAR L)) (CONC (CDR L)))
          (T (CONS (CAAR L)
                   (CONC (CONS (CDAR L) (CDR L)))))))
```

correspondent les trois concepts :

```
1- (CONC ()) → ()
2- (CONC (() . l)) → (PR (CONC l)
                         l' → l')                      ⓵
3- (CONC ((x . y) . l)) → (PR (CONC (y . l))
                              l' → (x . l'))
```

Par application de la règle d'équilibration sur chacun de ces concepts,
on obtient le graphe de dépendance :



qui exhibe 1 comme condition d'arrêt, et montre qu'aucun arc ne relie 3 à 1. Le chemin 3-1 est impossible. L'appel récursif de la troisième clause de CONC ne conduit pas directement à la condition d'arrêt de CONC.

La règle d'induction peut être appliquée sur 3, puis le résultat équilibré avec la règle 2 de CONC :

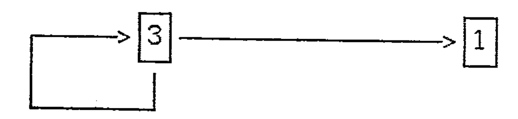
```
1- (CONC ()) → ()
2- (CONC (() . l)) → (PR (CONC l)
                         l' → l')                      ⓷
3- (CONC ((x1 ... xn) . l))
         → (PR (CONC l)
               l' → (x1 ... xn . l'))
```

Les résultats obtenus peuvent à nouveau être compactés en utilisant une règle d'***absorption***. La règle d'absorption réunit en un seul concept le résultat de l'induction et le concept avec lequel ce résultat est équilibré.

A la place de ⓷, on obtient :

```
1- (CONC ()) → ()
3- (CONC ((x1 ... xn) . l))
         → (PR (CONC l)
               l' → (x1 ... xn . l'))
```

Ce dernier exemple illustre particulièrement l'avantage de la règle d'absorption, puisque le graphe de dépendance de CONC est réduit à :



ce qui, par induction, équilibration et absorption, est compacté en :

(CONC *((x11 ... xn1) ... (x1m ... xnm))*)
→ *(x11 ... xn1 ... x1m ... xnm)*

AXIOMATISATION DES PRIMITIVES LISP

Le système CAN de compréhension automatique de programmes LISP est fondé sur les règles du calcul conceptuel: composition, équilibration, cercle-vicieux, induction constructrice.

Les règles du calcul conceptuel supposent une axiomatisation des domaines sur lesquels il est appliqué. Cette axiomatisation sert principalement la règle de composition, dont le rôle est de simplifier des expressions en concepts. Ces axiomatisations sont des systèmes de résolution d'équations.

Les axiomatisations requises par le calcul conceptuel procèdent de principes différents de ceux des axiomatisations classiques, destinées à être utilisées par des mathématiciens, ou utilisées par des systèmes de démonstration automatique. En particulier, elles ne sont pas basées sur le principe d'économie, visant à réduire au maximum le nombre d'axiomes de base. Le calcul conceptuel demande au contraire d'importants systèmes de déduction. Ces systèmes de déduction doivent minimiser les choix d'utilisation d'axiomes à chaque pas du processus de résolution d'équations.

En effet, le calcul conceptuel est censé obtenir ses résultats le plus rapidement possible, indépendamment des moyens utilisés. La résolution d'équations qu'utilise le calcul conceptuel ne doit pas s'engager dans des recherches arborescentes. Elle ne doit pas résoudre de problème. Elle constitue le pouvoir déductif de la compréhension de programmes. Ceci distingue la résolution d'équations dans le système CAN des systèmes axiomatiques qui cherchent à atteindre leurs buts avec le minimum de moyens.

La représentation de la fonction APPEND comme une fonction à deux arguments oblige à utiliser certains axiomes comme l'associativité et l'élément neutre à gauche et à droite. Ces axiomes et les déductions qu'ils permettent peuvent être ***intégrés*** dans un mode de représentation. De même peuvent y être intégrées les relations entre les fonctions CAR CDR CONS.

Nous utilisons les variables de ***séquence*** :

(APPEND L M) s'écrit (?L ?M)

où ?L et ?M ont pour domaine l'ensemble des séquences d'éléments, y compris la séquence vide.

(APPEND L NIL), par exemple, se représente (?L)

APPEND est défini ainsi :

(APPEND (?L) (?M)) = (?L ?M)

Cette représentation automatise l'utilisation de l'associativité-élément neutre de APPEND, et par l'utilisation de variables d'élément, distinguées par un "!", elle permet de définir CAR CDR CONS indépendamment les unes des autres:

```
(CAR  (!A ?B)) = !A
(CDR  (!A ?B)) = (?B)
(CONS !A (?B)) = (!A ?B)
```

Cette représentation uniforme des fonctions CAR CDR CONS APPEND à partir de filtres permet de réduire les systèmes de simplification et de résolution d'équations à la comparaison de filtres [GOOSSENS 78c].

L'utilisation de variables d'élément typées permet de définir conceptuellement les primitives d'un langage de programmation qui servent de connecteurs logiques. En LISP, par exemple, on peut définir:

```
(AND !X1 ... !Xn) = si (META-EVAL !X1)=NIL alors NIL
                    sinon
                       (AND !X2 ... !Xn)
(AND) = T

(OR !X1 ... !Xn) = si (META-EVAL !X1)=NIL
                      alors (OR !X2 ... !Xn)
                   sinon
                      (META-EVAL !X1)=!Y
                      !Y
(OR) = NIL

(IMPLIES !X !Y) = si (META-EVAL !X)=NIL alors T
                  sinon
                     (META-EVAL !X)=!Z
                     (META-EVAL !Y)

(NOT !X) = si (META-EVAL !X)=NIL alors T
           sinon NIL
```

(les variables soulignées ne peuvent être associées à la valeur LISP NIL. Du point de vue de la méta-évaluation, les définitions ci-dessus n'exhibent que le contrôle que les fonctions ont sur l'évaluation de leurs arguments. Les modifications qu'elles entrainent sur l'environnement courant ne sont pas considérées).

Les variables d'élément typées permettent également de définir les

prédicats de base (ex: ATOM, NUMBP, LISTP, STRING en LISP).

exemple:

```
(ATOM  !atom) = T
(ATOM  !~atom) = NIL
```

où !atom et !~atom sont respectivement une variable dont le domaine est l'ensemble des atomes LISP et une variable dont le domaine est l'ensemble des éléments non atomiques.

Ces accomodations permettent d'étendre les résultats théoriques de l'unification de chaines aux expressions formées à partir des primitives CAR CDR CONS NOT ATOM NUMBP STRING APPEND AND OR IMPLIES EQUAL.

La définition de APPEND:

```
(APPEND  (?A) (?B)) = (?A ?B)
```

améliore la définition classique:

```
(APPEND NIL L) = L
(APPEND (CONS X Y) L) = (CONS X (APPEND Y L))
```

qui ne permet pas de simplifier les formes :

```
(APPEND L NIL)
(EQUAL (APPEND L M) L)
(EQUAL (APPEND L M) NIL)
(EQUAL (APPEND L M) (APPEND L N))
(EQUAL (APPEND L L) (APPEND M M))
```

de même, la définition:

```
(REVERSE NIL) = NIL
(REVERSE (CONS X Y)) = (APPEND (REVERSE Y) (CONS X NIL))
```

ne permet pas de simplifier les formes :

```
(REVERSE (APPEND X (CONS Y NIL)))
(REVERSE (APPEND L M))
(REVERSE (REVERSE X))
(EQUAL (REVERSE X) X)
```

elle est améliorée par cette définition, utilisée par le système CAN:

```
(REVERSE (!X1 ... !Xn)) = (!Xn ... !X1)
```

qui s'écrit:

$$(\text{REVERSE } (< L_{i}^{n}\ !X_{i} >)) = (< L_{i}^{n}\ !X_{n-i+1} >)$$

Ce genre de définition demande à accepter un nouveau type de variable pour construire des filtres, les notations indicées.

Les notations indicées

Pour définir conceptuellement des fonctions courantes de manipulation de listes comme les fonctions LISP REVERSE, MEMBER, ASSOC, NTH, LENGTH, il faut utiliser un nouveau type de variable dont la syntaxe est:

$$<L^{p+}_{indice}\ \mathrm{FILTRES}>$$

Pour admettre ce nouveau type de variable, il faut également admettre des filtres numériques (p+ doit être un filtre numérique) et s'attendre à ce que les variables d'élément et de séquence soient indicées. Les filtres numériques sont exposés au paragraphe 8.2.6.

Cette notation indicée utilisée par le système de résolution d'équations défini au paragraphe 8.2.5.3, se définit algébriquement comme le type abstrait [GUTTAG 77, BERT 79] suivant:

type VSC (variable de séquence contrainte)

spécification syntaxique:

GEN : indice*filtre-numérique*filtre → VSC
PREMIER : VSC → filtre
COUPE-DROITE : VSC*filtre-numérique → VSC
COUPE-GAUCHE : VSC*filtre-numérique → VSC
ETENDRE : VSC*entier-positif → VSC
DECALAGE-CIRCULAIRE : VSC*entier-positif → VSC

sémantique:

$$(\text{PREMIER } (\text{GEN } i\ n\ p)) = p\Big|_{i}^{0}$$

$$(\text{COUPE-DROITE } (\text{GEN } i\ n\ p)\ m) = (\text{GEN } i\ n{-}m\ p\Big|_{i}^{i+m})$$

$$(\text{COUPE-GAUCHE } (\text{GEN } i\ n\ p)\ m) = (\text{GEN } i\ m\ p)$$

$$(\text{ETENDRE } (\text{GEN } i\ n\ p)\ a)$$

$$= (\text{GEN } i\ n/a\ p\Big|_{i}^{a*i}\ p\Big|_{i}^{a*(i-1)}\ \ldots\ p\Big|_{i}^{a*(i-(a-1))})$$

$$(\text{DECALAGE-CIRCULAIRE } (\text{GEN } i\ n\ p)\ a)$$

$$= \text{si } a<|p|$$

$$((\text{npremiers } a\ p)\Big|_{i}^{0}$$

$$(\text{GEN } i\ n{-}1\ (\text{saufpremiers } a\ p)$$

$$(\text{npremiers } a\ p)\Big|_{i}^{i+1})$$

$$(\text{saufpremiers } a\ p)\Big|_{i}^{n-1}))$$

NOMBRES

Les représentations conceptuelles de listes abstraites à l'aide de notations indicées exhibent des ***longueurs abstraites*** sous la forme de filtres numériques, c'est-à-dire à interpréter comme des classes d'entiers positifs.

Les manipulations d'entiers en LISP conduisent à des problèmes de compréhension qui reviennent le plus souvent à décider de la valeur d'une forme:

(relation expl ... expn)

où relation est une des relations de base en LISP: <, >, <=, >=, =. et où les expi sont des expressions numériques.

Du point de vue de la compréhension de programmes de traitement de listes, il faut particulièrement s'attacher au cas où les expi sont des expressions sur les entiers positifs, formés à partir de l'addition et la soustraction.
En choisissant les définitions conceptuelles:

```
(PLUS X1 ... Xn) = X1+ ... +Xn
(DIFFER X+Y Y) = X
(DIFFER X  X+Y+1) = 0
(<=  X  X+Y) = T
(<=  X+Y+1  X) = NIL
(>=  X  X+Y+1) = NIL
(>=  X+Y  X) = T
```

(< X X+Y+1) = T
(< X+Y X) = NIL
(> X X+Y) = NIL
(> X+Y+1 X) = T

Le problème auquel la résolution d'équations doit faire face se réduit à la résolution d'équations du type:

$$a1X1 + a2X2 + \ldots + anXn + an+1 = b1Y1 + \ldots + bmYm$$

où les Xi et les Yi sont des variables dont le domaine est celui des entiers positifs et où les ai et les bi sont des coefficients numériques constants, entiers et positifs.

Ce problème est un problème de résolution d'équations diophantiennes [STICKEL 75, HUET 78]. L'algorithme d'unification qu'utilise actuellement le système CAN n'est pas minimal. Il consiste en l'unifieur de séquences abstraites (règle 13, *SUIVANT*, *SUIVANT CONDITIONEL*) auquel s'ajoute une règle de simplification qui élimine les variables et constantes communes aux deux termes à unifier. Cet algorithme d'unification tient donc compte des propriétés de commutativité, associativité, élément neutre de +.

LE SYSTEME CAN

Le système CAN est un système de compréhension de programmes LISP, implémenté en VLISP [GREUSSAY 77, CHAILLOUX 80] sur un PDP KL-10. Il est basé sur les processus de méta-évaluation et de construction de significations exposés dans cette étude.
CAN fournit une description fonctionelle (sans effet de bord) des programmes qui lui sont soumis, et en simplifie la structure de contrôle. Cette description prend la forme d'un ensemble de concepts (éléments du calcul conceptuel défini au chapitre 7). Un programme est ainsi divisé, par l'analyse de cas opérée par méta-évaluation, en modules interdépendants. CAN construit et analyse le graphe de dépendance de cet ensemble de modules.
Cette description modulaire est ensuite compactée par induction, autant que le permettent les notations spéciales utilisées par CAN, et son pouvoir de résolution d'équations (chapitre 8).

En analysant leur graphe de dépendance, CAN est capable d'extraire des programmes qui lui sont soumis :

- des portions de code non utilisées
- des chemins impossibles
- des classes de données pour lesquelles le programme ne termine pas son calcul
- des redondances

Exemples :

Le programme MEMLONG suivant vérifie si deux listes ont la même longueur :

```
(DE MEMLONG (L M)
    (COND ((NULL L) (NULL M))
          ((NULL M) NIL)
          (T (MEMLONG (CDR L) (CDR M)))))
```

réponse de CAN :

```
=   (MEMLONG !L  !M ) →
=
=
=       Point-de-rupture [ !L  !M ]
=
=           (NIL !5 ) → NIL
=           ( !3 NIL) → NIL
=           (( (L<i ?N3+>  !9i )) ( (L<i ?N3+>  !14i ))) → T
=           (( (L<i ?N2+>  !9i ))
=            ( (L<i ?N2+>  !14i ) ?47 )) → NIL
=           (( (L<i ?N2+>  !9i ) ?54 )
=            ( (L<i ?N2+>  !14i ))) → NIL
```

Voici une définition incomplète de la fonction MEMBER :

```
?  (DE MEM (X L)
?     (IF (EQUAL X (CAR L))
?         L
?         (MEM X (CDR L))))
=  MEM
?
?  (CAN 'MEM)

=   (MEM !X  !L ) →
=
=
=       Point-de-rupture [ !X  !L ]
=
=           ( NIL
=           ( (L<i ?N2+>   !161i  / (NIL ≠ !161i ))) ) → NIL
=
=           (  !162
=           ( (L<i ?N3+>   !161i  / ( !162  ≠ !161i ))) )
=              → ( !162  NIL) : CALCUL INFINI
=
=           (  !166
=           ( (L<i ?N4+>  !161i /( !166 ≠ !161i )) !166 ?168 ) )
=              → ( !166  ?168 )
```

La classe de données pour lesquelles le calcul ne termine pas peut être décrite par : "les cas où X n'est pas un élément de la liste L" mais cette description exclut le cas particulier, que CAN décèle, où X a la valeur NIL. Dans ce cas, le calcul termine aprés que MEMBER ait parcouru toute la liste.

POUVOIR REPRESENTATIF

CAN utilise les résultats de la méta-évaluation d'un programme pour en construire une définition.
Pour mettre la méta-évaluation au service de la compréhension de programmes, nous avons dû généraliser les notions déjà connues d'évaluation symbolique, ou exécution symbolique. Cette généralisation a ainsi permis de tenir compte des fonctions LISP EVAL, SET et DF permettant un style de programmation particulièrement riche, puisque l'utilisateur peut définir et utiliser ses propres structures de contrôle, et écrire des programmes qui construisent et utilisent d'autres programmes.

Nous avons, par le calcul conceptuel, défini une base de représentation équivalente aux clauses de HORN [KOWALSKI 73], fondement du langage PROLOG [ROUSSEL 75], mais ouverte vers les extensions nécessaires à la compréhension de programmes :

- notations spécialisées évitant l'emploi de règles de simplification fréquemment utilisées

- définitions récursives compactables par induction et utilisables comme règles de simplification

Ces extensions nous ont permis d'appliquer le calcul conceptuel, dans le cadre de CAN, à la compréhension de programmes LISP. CAN est ainsi capable d'associer à un programme LISP une définition du calcul conceptuel, puis, si elle comporte des appels récursifs, de la compacter, c'est à dire de la débarasser de ses appels récursifs pour la rendre utilisable en tant que règle de simplification.

Exemple :

A partir de l'axiomatisation des fonctions CAR CDR CONS APPEND NULL :

```
(CAR (CONS X Y)) = X
(CDR (CONS X Y)) = Y

(NULL ()) = T
(NULL (CONS X Y)) = NIL

(APPEND () L) = ()
(APPEND (CONS X Y) L) = (CONS X (APPEND Y L))
```

et des règles d'inférence du calcul conceptuel, CAN peut, directement, prouver les théorèmes :

```
(APPEND X (APPEND X X)) = (APPEND (APPEND X X) X)
[(APPEND X X) = (APPEND Y Y)] => X = Y
```

mais aussi simplifier les expressions :

```
(APPEND X (APPEND Y Z)) = (APPEND (APPEND X W) Z)
                 en Y = W
(APPEND L M) = (APPEND L N)
```

```
                en M = N
(APPEND L L) = (APPEND M M)
                en L = M
```

POUVOIR DEDUCTIF

Le calcul conceptuel sans notations spécialisées (variables de séquence, notations indicées) offre un pouvoir représentatif équivalent à LISP. Cependant, il faut introduire des notations spécialisées pour développer le pouvoir déductif.
Le pouvoir déductif de CAN prend la forme de systèmes de résolution d'équations qui tiennent compte de la présente des notations spécialisées. C'est l'extension par ajout de notations spécialisées et de systèmes de résolution d'équations qui différencie CAN des systèmes de vérification et de compréhension de programmes fondés sur la logique des prédicats.
Cette voie est à rapprocher des tentatives de remplacement, dans un démonstrateur de théorèmes, d'axiomes fréquement utilisés (comme les propriétés d'associativité et de commutativité) par des procédures d'unification spécialisées [PLOTKIN 72, STICKEL 75, SIEKMANN 75]. Nous avons contribué à ces tentatives par l'ajout de notations spécialisées qui tiennent compte des propriétés particulières des fonctions LISP **REVERSE**, **MEMBER**, **ASSOC**, **NTH**, **LENGTH**, **LAST**. CAN est également capable, à partir d'une définition LISP, d'en dériver une description à base de ces notations spécialisées et donc d'utiliser son pouvoir déductif pour en comprendre les utilisations.

CAN peut être vu dans son état actuel, comme une première tentative d'adjoindre à tout système à trés longue durée de vie, une composante d'auto-compréhension nécessaire pour doter ce système d'une large capacité d'auto-maintenance.
Ceci peut donner à penser que l'action ponctuelle et dominante du programmeur actuel sur un système à construire ou à améliorer, fera place à trés long terme, à une interaction programmeur-système, où l'initiative du système pour les décisions concernant sa conception sera de plus en plus importante.

REFERENCES

[AUBIN 77]
AUBIN R. 1977
Strategies for mechanizing structural induction
Proc. 5th. IJCAI. MIT. Cambrige Mass. Aug 1977. pp 363-369.

[AUBIN 77b]
AUBIN R. 1977
Mechanizing structural induction
PH.D. Thesis Dept. of AI Univ. of Edinburgh, Scotland. 1977

[BALZER 77]
BALZER R. GOLDMAN N. WILE D. 1977
Meta-evaluation as a tool for program understanding
5th IJCAI, MIT Cambridge, August 1977

[BERT 79]
BERT D. 1979
Spécification algébrique des types abstraits et certification de programmes
AFCET Bulletin GROPLAN n.8 1979.

[BERZINS 78]
BERZINS V. 1978
Abstract model spécifications for data abstractions
PH.D.Thesis Draft. MIT Lab for comp. science.

[BORRAS 80]
BORRAS P. 1980
AIDE: un systeme d'évaluation symbolique du langage LISP
Rapport de stage. Univ. PARIS 6. Juin 80.

[BOYER 75a]
BOYER R. S. MOORE J. S. 1975
Proving theorems about LISP functions
JACM Vol. 22 n.1 pp. 129-144. 1975.

[BOYER 75b]
BOYER R.S. ELSPAS B. LEVITT K.N. 1975
SELECT--A formal system for testing and debugging programs by symbolic execution
Int. Conf. on Reliable Software, Los Angeles, April 1975, pp. 234-245.

REFERENCES

[BOYER 77]
BOYER R. S. MOORE J. S. 1977
A computer proof of the correctness of a simple optimizing compiler of expressions
Tech. rep. N00014-75-c-0816-SRI-4079. SRI International. Menlo Park. California. Jan 77.

[BURSTALL 69]
BURSTALL R.M. 1969
Proving properties of programs by structural induction
Computer Journal. Vol 12. n.1. Fev 69. pp 41-48.

[BURSTALL 72]
BURSTALL R.M. 1972
Some techniques for proving correctness of programs which alter data structures
Machine Intelligence 7. Michie ed. New York.

[BURSTALL 77]
BURSTALL R.M. DARLINGTON J. 1977
Some transformations for developping recursive programs
JACM Vol 24. n.1. Jan 77.

[CARTWRIGHT 76]
CARTWRIGHT R. 1976
A practical formal semantic definition and verification system for typed LISP
Memo-aim 296, Stanford Univ. Dept. of Comp. Science. Dec 76.

[CHAILLOUX 78a]
CHAILLOUX J. 1978
VLISP-8 un systeme LISP pour micro-processeur à mots de 8 bits
RT-21-78, Dept d'informatique. Univ. PARIS 8. Juillet 78.

[CHAILLOUX 78b]
CHAILLOUX J. 1978
a VLISP interpreter on the VCMC1 machine
LISP bulletin n.2, july 78.

[CHAILLOUX 78c]
CHAILLOUX J. 1978
VLISP-10 manuel de référence
RT-16-78, Dept d'informatique. Univ. PARIS 8. aout 78.

[CHAILLOUX 80]
CHAILLOUX J. 1980
Le modele VLISP: description, implementation, evaluation.
Thèse de 3ième cycle, Univ P. et M. CURIE.

REFERENCES

[CHARNIAK 72]
CHARNIAK E. 1972
Towards a model of children's story comprehension
AI-TR-266 MIT, AI-LAB, Dec. 1972.

[CHARNIAK 75]
CHARNIAK E. 1975
Organisation and inference in a frame-like system of common-sense knowledge
Working Paper 14. Institute for Semantic and Cognitive Studies. Castagnola, Switzerland 1975.

[CHEATHAM 79]
CHEATHAM T.E., HOLLOWAY G.H., TOWNLEY J.A. 1979
Symbolic evaluation and the analysis of programs
IEEE transactions on software engineering, vol SE-5, n.4, July 1979, pp 402-417.

[CHURCH 41]
CHURCH A. 1941
The calculi of λ-conversion
Annals of Math. studies n.6. Princeton Univ. Press. Princeton 1941.

[COOK 75]
COOK A. OPPEN D.C. 1975
An assertion language for data structures
2nd. ACM Symposium on principles of programming languages. Palo Alto California Jan. 75

[DURIEUX 78]
DURIEUX J.L. SALLE P. 1978
Application de la notion d'échappement à la description des instructions de saut
AFCET, Bulletin GROPLAN n.5 1978.

[FICKAS 79]
FICKAS S. BROOKS R. 1979
Recognition in a program understanding system
6th. IJCAI. Tokyo. Aug. 79. pp 266-268.

[FLOYD 67]
FLOYD R.W. 1967
Assigning meanings to programs
Math. Aspects of Comp. Science Proc. Symp. in Applied Math. 19, Providence (RI), Amer. Math. Soc. 1967, Schwartz ed.

[GERHART 76]
GERHART S. 1976
Proof theory of partial correctness verification systems
SIAM Journal Comput. Vol 5. n.3. Sept 76.

REFERENCES

[GOOSSENS 77]
GOOSSENS D. 1977
CAN: un systeme de méta-interprétation du langage LISP
Dept. d'Informatique, Univ. Paris 8-Vincennes, 1977.

[GOOSSENS 78a]
GOOSSENS D. 1978
A system for visual-like understanding of LISP programs
A.I.S.B. Conf. Hamburg, July 1978.

[GOOSSENS 78b]
GOOSSENS D. 1978
Compréhension visuelle de programmes controlée par méta-filtrage
Groplan: Bulletin de l'AFCET, Groupe Programmation et langages, 1978

[GOOSSENS 78c]
GOOSSENS D. 1978
L'unification au service de la compréhension
RT-14. Dept. d'informatique, Université de Vincennes.

[GOOSSENS 79]
GOOSSENS D. 1979
Meta-interpretation of recursive list-processing programs
6th. IJCAI. Tokyo. Aug 79.

[GREEN 75]
GREEN C. BARSTOW D. 1975
A hypothetical dialogue exhibiting a knowledge base for a program understanding system
Stanford A.I. Lab. Memo-Aim 258, Cpt Science Dept. Report n. Stancs-75-476.

[GREUSSAY 76]
GREUSSAY P. 1976
VLISP: structure et extension d'un systeme LISP pour mini-ordinateur
RT-16-76, UER Informatique et linguistique, Univ PARIS 8 Vincennes, Janvier 76.

[GREUSSAY 77]
GREUSSAY P. 1977
Contribution a la définition interprétative et à l'implémentation des λ-langages
Thèse, Université PARIS 7. Nov. 77.

[GREUSSAY 78]
GREUSSAY P. 1978
Le systeme VLISP-16
Ecole polytechnique. Décembre 78.

REFERENCES

[GREUSSAY 79]
GREUSSAY P. 1979
VLISP-11 manuel de référence
Université PARIS 8 Vincennes. 1979.

[GREUSSAY 80]
GREUSSAY P. 1980
Program understanding by reduction sets
7th. AISB. Amsterdam. 1980.

[GUTTAG 77]
GUTTAG J. 1977
Abstract data types and the development of data structures
Comm. of ACM. Vol. 20 n.6 Juin 1977. pp 396-404.

[HEWITT 69]
HEWITT C. 1969
PLANNER: A language for manipulating models and proving theorems in a robot
1st. IJCAI. Washington DC.

[HEWITT 72]
HEWITT C. 1972
Description and theoritical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot
MIT Revised Ph.D. Dissertation. AI-TR-258. April 72.

[HEWITT 75]
HEWITT C.E. SMITH B. 1975
Towards a programming apprentice
IEEE Trans. on soft. engineering, Vol. SE-1. pp. 26-45.

[HMELEVSKII 66]
HMELEVSKII J.I. 1966
Word équations without coefficients
Soviet Math Dokl. Vol 7 n.6 1966.

[HMELEVSKII 67]
HMELEVSKII J.I. 1967
Solution of word équations in three unknowns
Soviet Math Dokl. Vol 8 n.6 1967.

[HOARE 69]
HOARE C.A.R 1969
An axiomatic basis for computer programming
Comm. ACM, Vol 12, n.10, Oct. 1969. pp 576-583.

[HUET 75]
HUET G. 1975
A unification algorithm for typed λ-calculus
Theoritical Comp. Science 1, 1975.

REFERENCES

[HUET 77]
HUET G. 1977
Résolution d'équations dans les langages d'ordre 1, 2, ... omega
Thèse. Université PARIS 7. Sept. 77.

[HUET 78]
HUET G. 1978
An algorithm to generate the basis of solutions to homogeneous linear diophantine equations
Rapport de recherche n.274. Jan. 1978.

[IGARASHI 75]
IGARASHI LONDON LUCKHAM 1975
Automatic program verification I: A logical basis and its implementation
Acta Informatica, Vol. 4, pp. 145-182.

[KARR 76]
KARR M. 1976
Summation in finite terms
Mass. Computer Associates. Wakefield MA. Tech. Rep. Feb. 76

[KATZ 73]
KATZ S. MANNA Z. 1973
A heuristic approach to program verification
Proc. 3rd. Int. Conf. on AI. Palo-alto. California. 73. pp 500-512.

[KING 75]
KING J. 1975
A new approach to program testing
Int. Conf. on reliable software, April 1975, pp. 228-233.

[KING 76]
KING J. 1976
Symbolic execution and program testing
Comm. of ACM. Vol 19. n.7 July 76. pp 385-394

[KOWALSKI 73]
KOWALSKI R. 1973
Predicate logic as a programming language
Memo n.70, Dept. of Comp. Logic School of AI. Univ. of Edinburgh. Nov. 73.

[KUHNER 77]
KUHNER S. MATHIS C. RAULEFS P. SIEKMANN J.
Unification of idempotent functions
5th. IJCAI. MIT. Cambrige. Aug 77. p 528.

REFERENCES

[LENTIN 72]
LENTIN A. 1972
Equations dans les monoides libres
Gauthier Villars, Paris 1972.

[LIEVESEY 76]
LIEVESEY M. SIEKMANN J. 1976
Unification of bags and sets
Internal Report 3/76, Institut fur Informatic I, Univ. Karlsruhe.

[LISBUL 80]
LISP Bulletin n.2
P. Greussay et J. Laubsch eds. Univ. Paris 8, Dept. d'informatique. 1980.

[LISKOV 75]
LISKOV B.H. ZILLES S.
Specifications techniques for data abstractions
Proceedings of ACM Int. Conf. on Reliable Software. Los Angeles 1975.

[LISKOV 77]
LISKOV B.H. BERZINS V.
An appraisal of program specifications
MIT Lab. for Comp. Science. Memo 141-1 1977.

[LONDON 74]
LONDON R.L. MUSSER D.R. 1974
Application of a symbolic mathematical system to program verification
Proc. of ACM. 1974. pp 265-273.

[MACSYMA 75]
MACSYMA reference manual
Mathlab Group, Proj. MAC, MIT. Cambrige Mass.

[MAKANIN 77]
MAKANIN G.S. 1977
The problem of solvability of equations in a free semi-group
Soviet Math. Dokl. Vol 18 n.2 1977.

[MANNA 73]
MANNA Z. 1973
Inductive methods for proving properties of programs
Comm. of ACM. Vol 16. n.8. Aug 73. pp 491-502

[Mc. DERMOTT 72]
Mc. DERMOTT D.V. SUSSMANN G.J. 1972
The CONNIVER reference manual
AI-memo 259a. MIT AI-Lab. May 1972.

REFERENCES

[MEYER 72]
MEYER G.S. 1972
Infants in children stories. Towards a model of natural language comprehension
MIT AI-Lab. Aug. 72. Memo 265.

[MINSKY 75]
MINSKY M.
A framework for representing knowledge
In Ph. Winston ed. The psychology of computer vision. New York, Mc. Graw Hill 1975, pp 211-277.

[MOORE 73]
MOORE J. NEWELL A. 1973
How can MERLIN understand ?
Dept. of Comp. Science, Carnegie Mellon Univ. Pittsburgh Pennsylvannia, Nov. 73.

[OPPEN 78a]
OPPEN D.C. 1978
Reasonning about recursively defined data structures
5th Annual ACM symp. on principles of programming languages. jan 78. pp 151-157.

[OPPEN 78b]
OPPEN D.C. NELSON G. 1978
Simplification by cooperating decision procedures
Memo-aim-311 report n. STAN-CS-78-652. SAIL. April 78.

[OPPEN]
OPPEN D.C.
Tools for program analysis. Oppen.sli, Stanford Univ.

[PARK 69]
PARK D. 1969
Fixpoint induction and proof of program properties
Machine intelligence 5. Meltzer and Michie eds. Edinburgh Univ. Press. 1969. pp 59-78.

[PLOTKIN 72]
PLOTKIN G.D. 1972
Building-in equational theories
Mach. Int. 7, Meltzer Michie eds., 1972.

[RAPHAEL 71]
RAPHAEL B. 1971
The frame problem in problem-solving systems
Artificial Intelligence and Heuristic Programming. Findler and Meltzer eds. Edinburgh Univ. Press. 1971.

REFERENCES

[RAULEFS 78]
RAULEFS P. SIEKMANN J. SZABO P. UNVERICHT E. 1978
A short survey on the state of the art in matching and unification problems
Institut fur Informatik I. Univ. Karlsruhe.

[REDDY 75]
REDDY R. 1975 ed.
Speech recognition: invited papers presented at the IEEE Symposium
Academis Press, New York.

[REYNOLDS 72]
REYNOLDS J.C. 1972
Definitional interpreters for higher order programming languages
Proc. of 25th ACM National Conf. Boston, 1972.

[RICH 75]
RICH C. SCHROBE H.E. 1975
Understanding LISP programs: Towards a programming apprentice
Master's Thesis, EECS M.I.T.

[RICH 76]
RICH C. SCHROBE H.E. 1976
Initial report on a LISP programmer's apprentice
MIT AI-TR-354 Dec. 1976.

[RICH 79]
RICH C. SCHROBE H.E. WATERS R.C. 1979
Overwiew of the programmer's apprentice
6th. IJCAI. Tokyo. Aug 79. pp 827-828.

[ROBINET 78]
ROBINET B. 1978
Petit précis de λ-calcul
Implémentation et interprètation de LISP. Ecole IRIA. Toulouse. pp 15-24. Mars 1978.

[ROBINSON 65]
ROBINSON J. 1965
A machine-oriented logic based on the resolution principle
Journal of Assoc. for Comp. Machinery. Vol 12. n.1. Jan 1965. pp 23-41.

[ROUSSEL 75]
ROUSSEL P. 1975
PROLOG manuel de référence et d'utilisation
Groupe d'Intelligence Artificielle. UER Marseille-Luminy. Sept 75.

REFERENCES

[RUTH 74]
RUTH G.R. 1974
Analysis of algorithm implementation
MIT. MAC-TR-130.

[SCHANK 75a]
SCHANK R.C. GOLDMAN N. RIEGER C.J. RIESBECK C. 1975
MARGIE: Memory, analysis, response generation, and inference on english
Stanford Univ. 1973

[SCHANK 75b]
SCHANK R.C. 1975
Conceptual information processing
Noth Holland, Amsterdam.

[SCHANK 77]
SCHANK R.C. ABELSON R. 1977
Scripts, plans, goals and understanding
Lawrence Erlbaum Press, Hillsdale NJ.

[SCHUTZENBERGER 66]
SCHUTZENBERGER M.P. 1966
Quelques problemes combinatoires de la théorie des automates
Chap. II.1, Univ. Paris 6, Institut de Programmation, 1966.

[SELFRIGE 59]
SELFRIGE O. 1959
Pandemonium: a paradigm for learning
Symp. on the mechanization of thought processes. London HM Stationery Office.

[SETHI 77]
SETHI R. 1977
Semantics of computer programs: overview of language definition methods
Bell Laboratories, Murray Hill, New Jersey 07974.

[SHROBE 79]
SHROBE H.E. 1979
Dependency directed reasonning in the analysis of programs which modify complex data structures
6th. IJCAI. Tokyo. Aug 79. pp 829-835.

[SIEKMANN 75]
SIEKMANN J. 1975
String unification
Essex university, Memo CSM-7.

REFERENCES

[SIEKMANN 75b]
SIEKMANN J. LIVESEY M. 1975
Termination and decidability results for string unification
Essex university, CSM-12.

[SMITH 75]
SMITH B.C. HEWITT C. 1975
A plasma primer
MIT, AI-Lab, Draft. Sept 1975.

[STEELE 76]
STEELE G.L. SUSSMAN G.J.
Lambda, the ultimate imperative
AI-memo 353 MIT AI-Lab. March 76.

[STICKEL 75]
STICKEL 1975
A complete unification algorithm for associative-commutative functions
4th Ijcai, Tbilisi Georgia USSR. sept. 1975.

[TENNENT 76]
TENNENT R.D. 1976
The denotational semantics of programming languages
Comm. of ACM, Aug 76, Vol. 19, n.8, pp 528-535.

[VAN EMDEN 76]
VAN EMDEN et KOWALSKI R.A. 1976
The semantics of predicate logic as a programming language
JACM. Vol 23. n.4. Oct 1976. pp 733-742.

[WALDINGER 74]
WALDINGER R. LEVITT K.N. 1974
Reasonning about programs
Art. Int. Vol 5 n.3, North-holland Amsterdam pp 235-316.

[WALTZ 79]
WALTZ D.L. BOGESS L.
Visual analog representations for natural language understanding
6th. IJCAI, Tokyo. pp 926-934 Aug.1979.

[WARREN 77]
WARREN H.D. 1977
Implementing PROLOG
Vols. 1 et 2, DAI Research Report n.40, Univ. of Edinburgh, Dept of AI, 1977.

[WATERS 78]
WATERS R.C. 1978
Automatic analysis of the logical structure of programs
Ph.D. Thesis. MIT. AI-Lab. 1978.

REFERENCES

[WATERS 79]
WATERS R.C. 1979
A method for automatically analysing programs
6th. IJCAI. Tokyo. Aug. 1979. pp 935-941.

[WEGBREIT 73]
WEGBREIT B. 1973
Heuristic methods for mechanically deriving inductive assertions
Proc. 3rd. IJCAI, Aug 1973, pp 524-536.

[WEGBREIT 76]
WEGBREIT B. SPITZEN J.M. 1976
Proving properties of complex data structures
Journal of the Assoc. for Computing Machinery, Vol 23 n.2, April 76 pp 389-396.

[WERTZ 78a]
WERTZ H. 1978
Correction automatique de programmes LISP
Journées SESORI sur la programmation. St. REMY de Provence. Mai 78.

[WERTZ 78b]
WERTZ H. 1978 *Un systeme de compréhension, de programmes incorrects*
Proc. 3ème colloque sur la programmation, Paris B. Robinet éd. pp 31-49.

[WERTZ 78c]
WERTZ H. 1978
Un systeme de compréhension, d'amélioration, et de correction de programmes incorrects
Thèse de 3ème cycle, Univ. P. et M. CURIE.

[WINOGRAD 73]
WINOGRAD T. 1973
Breaking the complexity barrier (again)
Proc. of the ACM. SIGIR-SIGPLAN interface meeting. Nov 73.

[YONEZAWA 75]
YONEZAWA Akinori 1975
meta-evaluation of actors with side-effects
Working paper 101, MIT AI-Lab, june 1975.

[YONEZAWA 76]
YONEZAWA A. HEWITT C. 1976 *Symbolic evaluation using conceptual representations for programs with side-effects*
M.I.T., A.I. Lab., AI-Memo 399. Dec. 76.

REFERENCES

[YONEZAWA 77]
YONEZAWA A. 1977
Specification and verification techniques for parallel programs based on message passing semantics
PH.D.Thesis, MIT Lab for Comp. Science, MIT-LCS-TR-191. Dec. 1977.